# Functional Programming and the Scala Language

## Lecture 4

**Eugene Zouev**
Innopolis University
Spring Semester 2018

# To Remind:

- Function literals; closures
- Partially-applied functions & currying
- By-name parameters
- Tuples & traits
- Currying & new control structures

**Today**:

- Companion objects
- Basic control structures
- Pattern matching

All code examples are taken from the book of M.Odersky Programming in Scala

# Companion Objects

## Java/C# etc.

```
class Example
{
   public int m1;
   private float m2;
   public static int m3;
}
```

## Scala approach

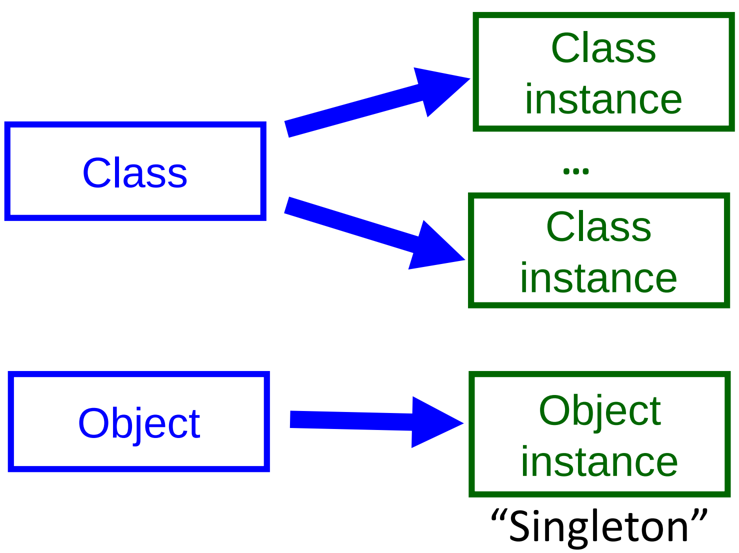
Companion class

```
class Example
{
   public int m1;
   private float m2;
}
```

Same name

```
object Example
{
   public int m3;
}
```

Companion object




- Class instances & singleton can access each others' private members
- Both class & object should be placed to the same source file

# Basic Control Structures

Scala has "typical" set of control structures:

- Conditional (`if`)
- Loops (`while`, `do`, `for`)
- Exception handling (`try`)

However…

Almost all of them have a bit different semantics, and…

There are some new constructs:

- `match`

The first difference: almost all of them **return values**, i.e., they are **expressions**.

# Control Structures: Conditional

```
var filename : String

if ( !args.isEmpty )
  filename = args(0)
else
  filename = "default.txt"
```

Usual semantics; imperative style

Notice the difference

```
val filename =
  if ( !args.isEmpty )
    args(0)
  else
    "default.txt"
```

Usual semantics; functional style

# Control Structures: Conditional

```
var filename : String

if ( !args.isEmpty )
  filename = args(0)
else
  filename = "default.txt"
```

```
val filename =
  if ( !args.isEmpty )
    args(0)
  else
    "default.txt"
```

Advantages for using **val** over **var**:

- Informs readers that the variable won't change later.
- **Equational reasoning**: the introduced variable is equal to the expression that computes it (assuming that expression has no side effects).

```
println(if (!args.isEmpty) args(0) else "default.txt")
```

M.Odersky: Look for opportunities to use **val**s. They can make your code both easier to read and easier to refactor.

# Control Structures: While Loops

```
def gcdLoop(x: Long, y: Long): Long =
{
  var a = x
  var b = y
  while (a != 0) {
    val temp = a
    a = b % a
    b = temp
  }
  b
}
```

Greater common denominator algorithm: while loop with usual semantics (imperative style)

The difference: these constructs are **not expressions**, but **loops**. They do not produce results. Therefore, they are typically **left out of pure functional languages**.

```
var line = ""
do {
   line = readLine()
   println("Read: "+ line)
} while (line != "")
```

Do-while loop with usual semantics (imperative style)

# Control Structures: While Loops

```
var line = ""
do {
   line = readLine()
   println("Read: "+ line)
} while (line != "")
```

What does it mean exactly "loops do not produce values" in Scala?

- Actually, they **do produce** the value.
  The type of the resulting value is `Unit`.

- The only value of the type `Unit` exists.
  It's called *unit value*, and it is written as `()`.

# Control Structures: While Loops

Actually, not about control structures…

The typical idiom for C/C++/Java

```
var line = "“
while ((line = readLine()) != "")
  println("Read: "+ line)
```

**This doesn't work in Scala!**

One more difference:

Assignments to **var**s don't produce values!

(or, which is the same, they produce `()`)

```
...(line = readLine()) != "")...
```

Always returns `()` of type `Unit`

Constant of type `String`

=> **Comparison is always true**

# Control Structures: While Loops

```
def gcdLoop(x: Long, y: Long): Long =
{
  var a = x
  var b = y
  while (a != 0) {
    val temp = a
    a = b % a
    b = temp
  }
  b
}
```

Greater common denominator algorithm: while loop with usual semantics (imperative style)

Notice the absence of `return` statement

Functional equivalent to gcd algorithm (using recursion)

```
def gcd(x: Long, y: Long): Long =
  if (y == 0) x else gcd(y, x % y)
```

# Control Structures: For Loops

M. Odersky:
Scala's **for expression** is a Swiss army knife of iteration

Notice "for ***expression***" in the statement

For loops can express a wide variety of iterations (including iterations over arbitrary collections) and even **produce new collections**.

Very first example (not common in Scala):

```
val filesHere = (new java.io.File(".")).listFiles

for (i <- 0 to filesHere.length - 1)
   println(filesHere(i))
```

# Control Structures: For Loops

Very first example (not common in Scala):

`Array[File]`

```
val filesHere = (new java.io.File(".")).listFiles

for (i <- 0 to filesHere.length - 1)
   println(filesHere(i))
```

The construct within parentheses is called **generator**
*(do you remember infix operator syntax?)*

**Generator** is any expression whose type contains `foreach` method

`i` is **val-variable** that is created on each iteration and gets initialized by the new value produced by generator. The type of `i` is inferred from the type of the `to` operator.

Notice: `i` is necessary only for indexing array elements…

# Control Structures: For Loops

A better solution (more in Scala style):

Array[File]

```
val filesHere = (new java.io.File(".")).listFiles

for (file <- filesHere)
   println(file)
```

Here, we iterate through the elements of filesHere, instead of iterating through integers then used to get elements.

files is of type File; while passing to println it gets implicitly converted to type String by applying the File's function toString.

**For loops can be applied to any Scala collection**

# Control Structures: For Loops

## Filtering

Common form:

```
for (v <- generator if condition)
    actions
```

Example:

```
val filesHere = (new java.io.File(".")).listFiles

for (file <- filesHere
                if file.getName.endsWith(".scala"))
    println(file)
```

The code prints only those files and directories in the current directory whose names end with "`.scala`".

# Control Structures: For Loops

## Multiple Filters

Notice the absence of parentheses around conditions

Example:

```
val filesHere = (new java.io.File(".")).listFiles

for (file <- filesHere
                if file.isFile
                if file.getName.endsWith(".scala"))
   println(file)
```

The code prints only those files (**but not subdirectories**) in the current directory whose names end with "`.scala`".

Notice that this for loop **doesn't produce a value**: it is used only for performing side effect (printing)

# Control Structures: For Loops

## Nested iterations

```
val filesHere = (new java.io.File(".")).listFiles

def fileLines(file: java.io.File) =
  scala.io.Source.fromFile(file).getLines().toList

def grep(pattern: String) =
  for( file <- filesHere
         if file.getName.endsWith(".scala");
       line <- fileLines(file)
         if line.trim.matches(pattern) )
    println(file +": "+ line.trim)

grep(".*gcd.*")
```

Notice semicolon separating nested loops

- The first (outer) loop selects files in the current directory whose names end with "`.scala`".
- The second (inner) loop takes the file selected on each iteration of the outer loop and selects lines that match the pattern from the `grep`'s parameter.
- The loop body prints lines that were selected.

# Control Structures: For Loops

## Producing the new collection

```
for Clauses yield Body
```

Example:

```
val filesHere = (new java.io.File(".")).listFiles

def scalaFiles =
    for { file <- filesHere
          if file.getName.endsWith(".scala")
    } yield file
```

Array[File]

Notice curly braces instead of parentheses

- After each iteration, the value of the body is stored.
- The overall result of the for-expression is **a new collection** composed of values produced on all iterations.
- The kind of collection is the same as the kind of the source collection. In the example, it's `Array[File]`.

# Match Expression

No `switch` statement - `match` expression instead

Alternatives can be **of any type** (strings in the example), and can be **non-constants**.

Clear semantics; imperative style

```
val firstArg = if (args.length>0) args(0) else ""

firstArg match {
  case "salt"  => println("pepper")
  case "chips" => println("salsa")
  case "eggs"  => println("bacon")
  case _       => println("huh?")
}
```

The default case: notice use of underscore that is popular in Scala ☺

Notice the absence of "break" in branches

# Match Expression

No `switch` statement - `match` expression instead

Clear semantics;
more functional style

```
val firstArg = if (args.length>0) args(0) else ""

val friend = firstArg match {
  case "salt"  => "pepper"
  case "chips" => "salsa"
  case "eggs"  => "bacon"
  case _       => "huh?"
}
println(friend)
```

# More about match

## Guards in cases

```
val ch: Char
  ...
val sign = ch match {
  case '+' => 1
  case '-' => -1
  case _   => 0
}
```

```
val ch: Char
  ...
val sign = ch match {
  case '+' => 1
  case '-' => -1
  case '0' =>
  case '1' =>
   ...
  case '9' => Character.digit(ch,10)
  case _   => 0
}
```

```
val ch: Char
  ...
val sign = ch match {
  case '+' => 1
  case '-' => -1
  case _   if Character.isDigit(ch) =>
                     Character.digit(ch,10)
  case _   => 0
}
```

# More about match

## Variables in cases

```
val ch: Char
  ...
val sign = ch match {
  case '+' => 1
  case '-' => -1
  case _   if Character.isDigit(ch) =>
                  Character.digit(ch,10)
  case _   => 0
}
```

```
var str: String
  ...
val sym = str(i) match {
  case '+' => 1
  case '-' => -1
  case ch  => Character.digit(ch,10)
}
```

# More about match

## Types in cases

```
val obj = ...
  ...
val res = obj match {
  case x: Int    => x
  case s: String => Integer.parseInt(s)
  case _: BigInt => Int.MaxValue
  case _         => 0
}
```

**Typed pattern**: generalization of Java's `instanceOf`

- If type of `obj` is `Int` then `match` returns it as it is;
- Otherwise, if type of `obj` is `String` then the binary representation of the integer from the string is returned;
- Otherwise, if type of `obj` is `BigInt` then the maximal integer value is returned;
- Otherwise, `match` returns integer zero.

# More about match

## Types in cases

Any is the root class
in Scala type hierarchy

```
def generalSize(x: Any) =
    x match {
        case s: String   => s.length
        case m: Map[_,_] => m.size
        case _           => -1
    }
```

Matches any object of class Map with arbitrary types of keys & values

Examples:

```
generalSize("abc")
generalSize(Map(1->'a',2->'b',3->'c'))
generalSize(math.Pi)
```

# More about match

## Collections in cases

### Tuples

```
val roots = QRoots(a,b,c)
  ...
val res = roots match {
  case (_, 0.0)        => "The 2nd root is zero"
  case (x, y) if x==y => "The single root"
  case (_, _)          => "Two different roots"
}
```

See L.2: `QRoots` returns the tuple of two **`Doubles`**

Simple access to parts (elements) of collections: **destructuring**

### Arrays

```
anArray match {
  case Array(0)              => "array with 0"
  case Array(x, y) if x==y => "array with two equal elements"
  case Array(_, _)           => "array with two elements"
}
```

What happens if **`case _`** branch is absent?

# Patterns in...

## ...Declarations

```
val x = 1
val y = 2
```

```
val (x, y) = (1, 2)
```

```
val (r1, r2) = QRoots(a,b,c)
```

## ...For-expressions

```
val map: Map[Int,String]
...
for ((k, v) <- map)
  println(k + "->" + v);
```

Destructures each map element into key&value pair and initializes k & v vals by the pair elements

```
for ((k, "") <- map)
  println(k);
```

Pattern (k,"") matches all map elements with empty values.

```
for ((k, v) <- map if v!="")
  println(k + "->" + v);
```

Prints all map elements with non-empty values.

# match: Pattern Matching in Full

Simplified expression grammar

```
Expression : Variable
           | Number
           | UnaryOperator Expression
           | Expression BinaryOperator Expression
```

How the grammar could be represented programmatically:

```
abstract class Expr
     class Var(name: String)
     class Number(num: Double)
     class UnOp(op: String, arg: Expr)
     class BinOp(op: String, left: Expr, right: Expr)
```

Expression example

a*(b+2)

```
new BinOp("*",
          new Var("a"),
          new BinOp("+",
                    new Var("b"),
                    new Number(2)))
```

# match: Pattern Matching in Full

```
new BinOp("*",
          new Var("a"),
          new BinOp("+",
                    new Var("b"),
                    new Number(2)))
```

Much better!

```
BinOp("*",
      Var("a"),
      BinOp("+",
            Var("b"),
            Number(2)))
```

a*(b+2)

For that, classes should be declared as follows:

```
abstract class Expr
case class Var(name: String)
case class Number(num: Double)
case class UnOp(op: String, arg: Expr)
case class BinOp(op: String, left: Expr, right: Expr)
```

1. Instances of case-classes can be created by their simple names, without new keyword: `val v = Number(1.0)`.
2. Class parameters become instance members (fields) with `val` specifier.
3. `toString`, `hashCode` & `equals` (==) methods are automatically added to case classes: `op.right == Number(2)`

# match: Pattern Matching in Full

The task: to implement **expression simplification** using a few obvious rules

Transformation rules ("algebra"):

-(-e) ➔ e //double negation
e + 0 ➔ e // adding zero
e * 1 ➔ e // multiplication by one

where *e* is an expression

```
UnOp("-", UnOp("-", e))       => e
BinOp("+", e, Number("0")) => e
BinOp("*", e, Number("1")) => e
BinOp("*", e, Number("0")) => e
```

# match: Pattern Matching in Full

Transformation rules ("algebra"):

-(-e) ➔ e   //double negation
e + 0 ➔ e   // adding zero
e * 1 ➔ e   // multiplication by one

where e is an expression

```
UnOp("-", UnOp("-", e))     => e
BinOp("+", e, Number("0")) => e
BinOp("*", e, Number("1")) => e
BinOp("*", e, Number("0")) => e
```

```
def simplify(expr: Expr): Expr =
  expr match {
    case UnOp("-", UnOp("-",expr))   => expr
    case BinOp("+",expr,Number("0")) => expr
    case BinOp("*",expr,Number("1")) => expr
    case _                           => expr
}
```